Kotlin - Quick Guide

Kotlin - Overview

Kotlin is a new open source programming language like Java, JavaScript, etc. It is a high level strongly statically typed language that combines functional and technical part in a same place. Currently, Kotlin targets Java and JavaScript. It runs on JVM.

Kotlin is influenced by other programming languages such as Java, Scala, Groovy, Gosu, etc. The syntax of Kotlin may not be exactly similar to JAVA, however, internally Kotlin is reliant on the existing Java Class library to produce wonderful results for the programmers. Kotlin provides interoperability, code safety, and clarity to the developers around the world.

Advantages and Disadvantages

Following are some of the advantages of using Kotlin for your application development.

Easy Language − Kotlin is a functional language and very easy to learn. The syntax is pretty much similar to Java, hence it is very easy to remember. Kotlin is more expressive, which makes your code more readable and understandable.

Concise − Kotlin is based on JVM and it is a functional language. Thus, it reduce lots of boiler plate code used in other programming languages.

Runtime and Performance − Better performance and small runtime.

Interoperability − Kotlin is mature enough to build an interoperable application in a less complex manner.

Brand New − Kotlin is a brand new language that gives developers a fresh start. It is not a replacement of Java, though it is developed over JVM. It is accepted as the first official language of android development. Kotlin can be defined as - Kotlin = JAVA + extra updated new features.

Following are some of the disadvantages of Kotlin.

Namespace declaration − Kotlin allows developers to declare the functions at the top level. However, whenever the same function is declared in many places of your application, then it is hard to understand which function is being called.

No Static Declaration − Kotlin does not have usual static handling modifier like Java, which can cause some problem to the conventional Java developer.

Kotlin - Environment Setup

However, if you still want to use Kotlin offline in your local system, then you need to execute the following steps to configure your local workspace.

Step 1 − Java 8 installation.

Kotlin runs on JVM, hence. it is really necessary to use JDK 8 for your local Kotlin development. Please refer to the official website of oracle to download and install JDK 8 or an above version. You might have to set the environment variable for JAVA such that it can work properly. To verify your installation in Windows operating system, hit “java –version” in the command prompt and as an output it will show you the java version installed in your system.

Step 2 − IDE installation.

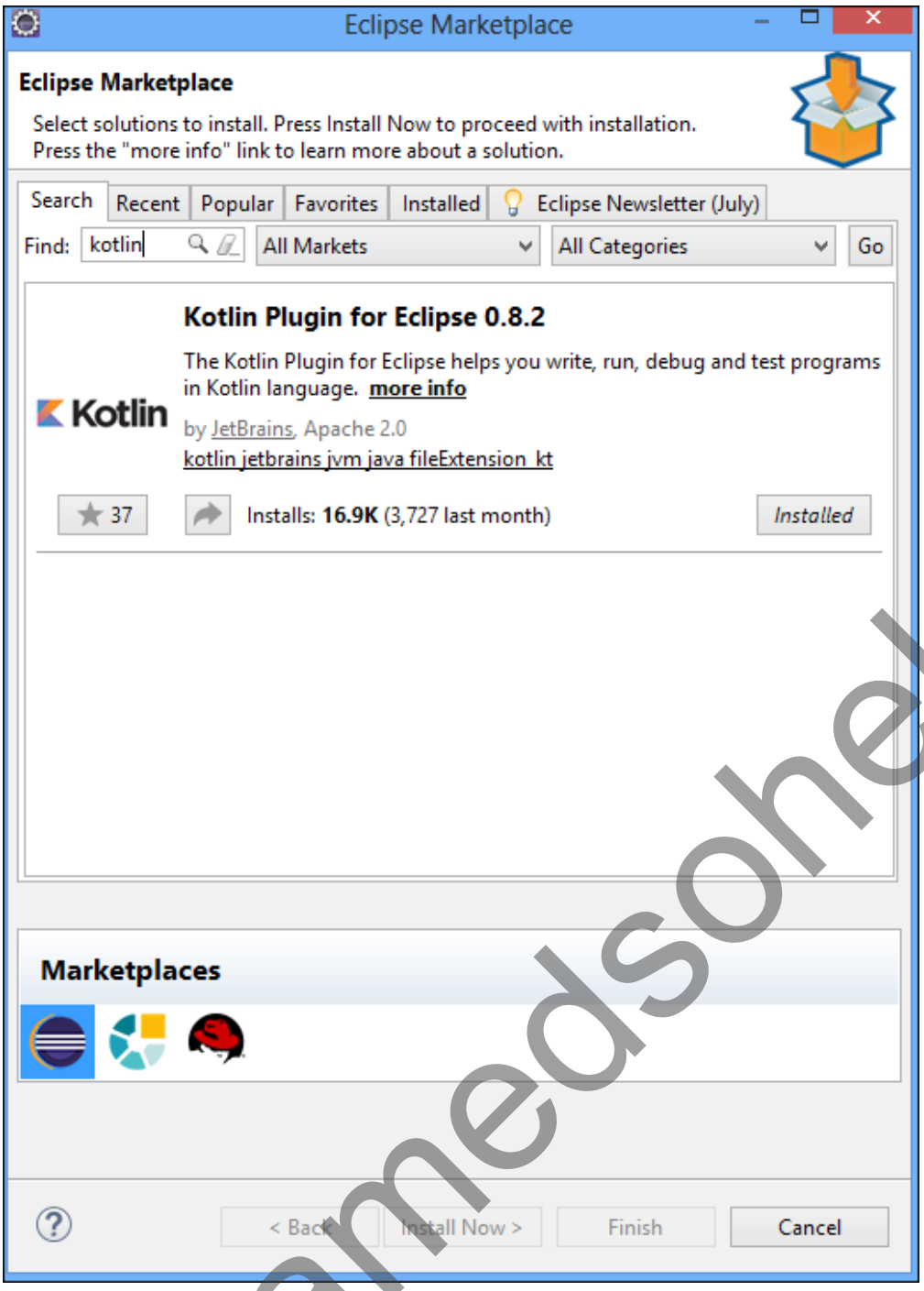
There are a number of IDE available over the internet. You can use any of your choice. You can find the download link of different IDE in the following table.

IDE Name	Installation Link
NetBeans	https://netbeans.org/downloads/
Eclipse	https://www.eclipse.org/downloads/
Intellij	https://www.jetbrains.com/idea/download/#section = windows

It is always recommended to use the recent software version to drag out maximum facility from it.

Step 3 − Configuring Eclipse.

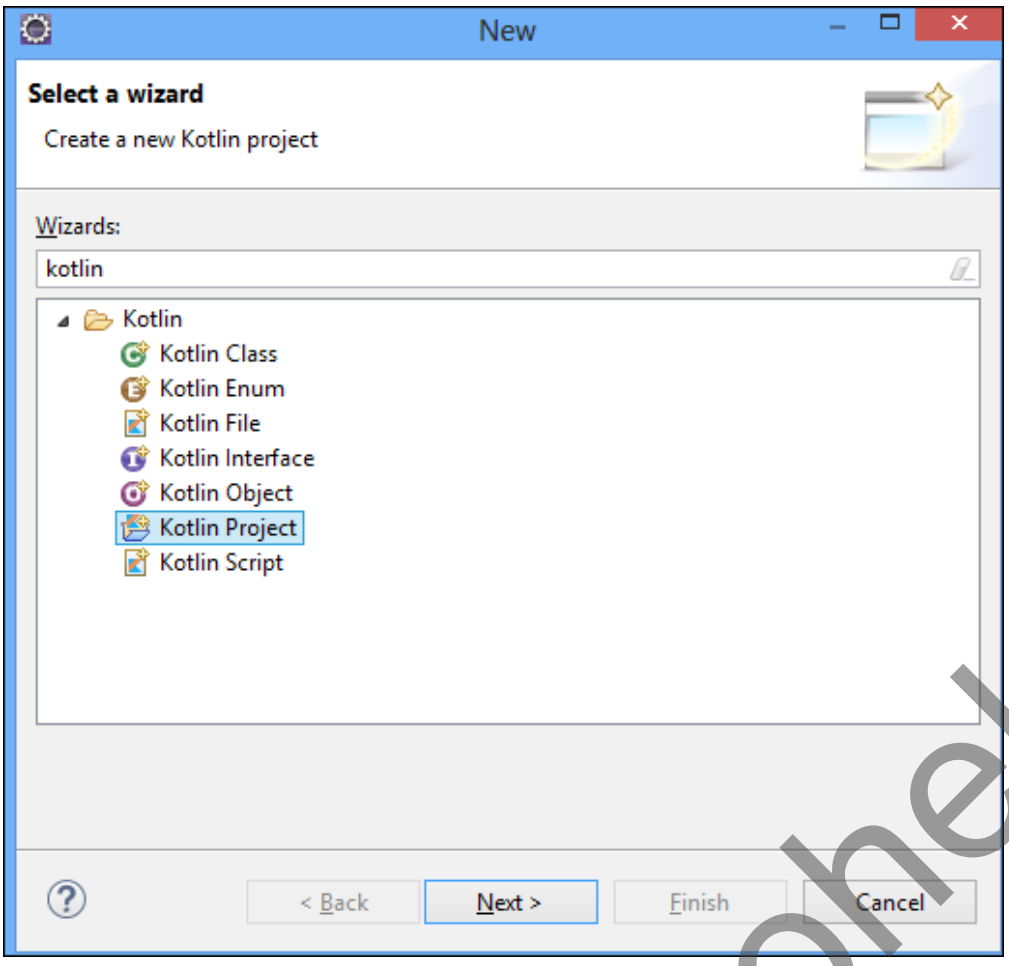
Open Eclipse and go to “Eclipse Market Place”. You will find the following screen.



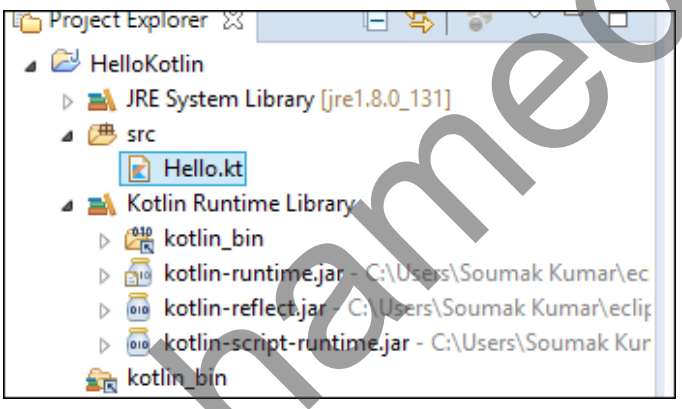
Search for Kotlin in the search box and install the same in your local system. It might take some time depending on the internet speed. You may have to restart your Eclipse, once it is successfully installed.

Step 4 − Kotlin Project.

Once Eclipse is successfully restarted and Kotlin is installed, you will be able to create a Kotlin project on the fly. Go to **File** → **New** → **Others** and select “Kotlin project” from the list.



Once the project setup is done, you can create a Kotlin file under “SRC” folder. Left-click on the “Src” folder and hit “new”. You will get an option for Kotlin file, otherwise you may have to search from the “others”. Once the new file is created, your project directory will be look like the following.



Your development environment is ready now. Go ahead and add the following piece of code in the “Hello.kt” file.

Live Demo

```
fun main(args: Array<String>) {
   println("Hello, World!")
}
```

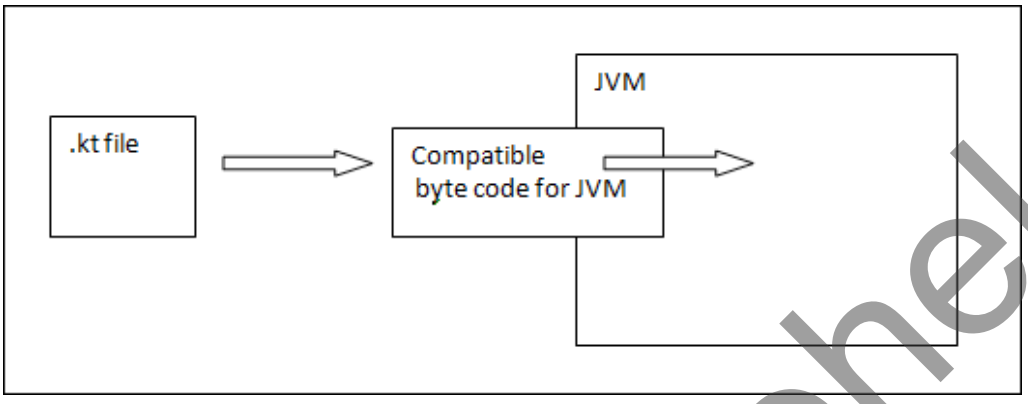
Run it as a Kotlin application and see the output in the console as shown in the following screenshot. For better understanding and availability, we will be using our coding ground tool.

```
Hello, World!
```

Kotlin - Architecture

Kotlin is a programming language and has its own architecture to allocate memory and produce a quality output to the end user. Following are the different scenarios where Kotlin compiler will work differently, whenever it is targeting different other kind of languages such as Java and JavaScript.

Kotlin compiler creates a byte code and that byte code can run on the JVM, which is exactly equal to the byte code generated by the Java **.class** file. Whenever two byte coded file runs on the JVM, they can communicate with each other and this is how an interoperable feature is established in Kotlin for Java.



Whenever Kotlin targets JavaScript, the Kotlin compiler converts the **.kt** file into ES5.1 and generates a compatible code for JavaScript. Kotlin compiler is capable of creating platform basis compatible codes via LLVM.

Kotlin - Basic Types

In this chapter, we will learn about the basic data types available in Kotlin programming language.

Numbers

The representation of numbers in Kotlin is pretty similar to Java, however, Kotlin does not allow internal conversion of different data types. Following table lists different variable lengths for different numbers.

Type	Size
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

In the following example, we will see how Kotlin works with different data types. Please enter the following set of code in our coding ground.

```
fun main(args: Array<String>) {
   val a: Int = 10000
   val d: Double = 100.00
   val f: Float = 100.00f
   val l: Long = 1000000004
   val s: Short = 10
   val b: Byte = 1

   println("Your Int Value is "+a);
   println("Your Double  Value is "+d);
   println("Your Float Value is "+f);
   println("Your Long Value is "+l);
   println("Your Short Value is "+s);
   println("Your Byte Value is "+b);
}
```

When you run the above piece of code in the coding ground, it will generate the following output in the web console.

```
Your Int Value is 10000
Your Double Value is 100.0
Your Float Value is 100.0
Your Long Value is 1000000004
Your Short Value is 10
Your Byte Value is 1
```

Characters

Kotlin represents character using **char**. Character should be declared in a single quote like **'c'**. Please enter the following code in our coding ground and see how Kotlin interprets the character

variable. Character variable cannot be declared like number variables. Kotlin variable can be declared in two ways - one using **“var”** and another using **“val”**.

```
fun main(args: Array<String>) {
   val letter: Char    // defining a variable
   letter = 'A'        // Assigning a value to it
   println("$letter")
}
```

The above piece of code will yield the following output in the browser output window.

```
A
```

Boolean

Boolean is very simple like other programming languages. We have only two values for Boolean – either true or false. In the following example, we will see how Kotlin interprets Boolean.

```
fun main(args: Array<String>) {
   val letter: Boolean   // defining a variable
   letter = true         // Assinging a value to it
   println("Your character value is "+"$letter")
}
```

The above piece of code will yield the following output in the browser.

```
Your character value is true
```

Strings

Strings are character arrays. Like Java, they are immutable in nature. We have two kinds of string available in Kotlin - one is called **raw String** and another is called **escaped String**. In the following example, we will make use of these strings.

```
fun main(args: Array<String>) {
   var rawString :String  = "I am Raw String!"
   val escapedString : String  = "I am escaped String!\n"

   println("Hello!"+escapedString)
   println("Hey!!"+rawString)
}
```

The above example of escaped String allows to provide extra line space after the first print statement. Following will be the output in the browser.

```
Hello!I am escaped String!

Hey!!I am Raw String!
```

Arrays

Arrays are a collection of homogeneous data. Like Java, Kotlin supports arrays of different data types. In the following example, we will make use of different arrays.

```
fun main(args: Array<String>) {
   val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)
   println("Hey!! I am array Example"+numbers[2])
}
```

The above piece of code yields the following output. The indexing of the array is similar to other programming languages. Here, we are searching for a second index, whose value is “3”.

```
Hey!! I am array Example3
```

Collections

Collection is a very important part of the data structure, which makes the software development easy for engineers. Kotlin has two types of collection - one is **immutable collection** (which means lists, maps and sets that cannot be editable) and another is **mutable collection** (this type of collection is editable). It is very important to keep in mind the type of collection used in your application, as Kotlin system does not represent any specific difference in them.

```
fun main(args: Array<String>) {
   val numbers: MutableList<Int> = mutableListOf(1, 2, 3) //mutable List
   val readOnlyView: List<Int> = numbers                  // immutable list
   println("my mutable list--"+numbers)        // prints "[1, 2, 3]"
   numbers.add(4)
   println("my mutable list after addition --"+numbers)     // prints "[1, 2, 3, 4
   println(readOnlyView)
   readOnlyView.clear()    // ⇒ does not compile
// gives error
}
```

The above piece of code will yield the following output in the browser. It gives an error when we try to clear the mutable list of collection.

```
main.kt:9:18: error: unresolved reference: clear
readOnlyView.clear() // -> does not compile
^
```

In collection, Kotlin provides some useful methods such as **first(), last(), filter()**, etc. All these methods are self-descriptive and easy to implement. Moreover, Kotlin follows the same structure such as Java while implementing collection. You are free to implement any collection of your choice such as Map and Set.

In the following example, we have implemented Map and Set using different built-in methods.

```
fun main(args: Array<String>) {
   val items = listOf(1, 2, 3, 4)
   println("First Element of our list----"+items.first())
   println("Last Element of our list----"+items.last())
   println("Even Numbers of our List----"+items.
      filter { it % 2 = = 0 })   // returns [2, 4]

   val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
   println(readWriteMap["foo"])  // prints "1"

   val strings = hashSetOf("a", "b", "c", "c")
   println("My Set Values are"+strings)
}
```

The above piece of code yields the following output in the browser.

```
First Element of our list----1
Last Element of our list----4
Even Numbers of our List----[2, 4]
1
My Set Values are[a, b, c]
```

Ranges

Ranges is another unique characteristic of Kotlin. Like Haskell, it provides an operator that helps you iterate through a range. Internally, it is implemented using **rangeTo()** and its operator form is **(..)**.

In the following example, we will see how Kotlin interprets this range operator.

```
fun main(args: Array<String>) {
   val i:Int  = 2
   for (j in 1..4)
   print(j) // prints "1234"

   if (i in 1..10) { // equivalent of 1 < = i && i < = 10
      println("we found your number --"+i)
```

```
    }
}
```

The above piece of code yields the following output in the browser.

```
1234we found your number --2
```

Kotlin - Control Flow

In the previous chapter we have learned about different types of data types available in Kotlin system. In this chapter, we will discuss different types of control flow mechanism available in the Kotlin.

If - Else

Kotlin is a functional language hence like every functional language in Kotlin **“if”** is an expression, it is not a keyword. The expression **“if”** will return a value whenever necessary. Like other programming language, **“if-else”** block is used as an initial conditional checking operator. In the following example, we will compare two variables and provide the required output accordingly.

```
fun main(args: Array<String>) {
    val a:Int = 5
    val b:Int = 2
    var max: Int

    if (a > b) {
        max = a
    } else {
        max = b
    }
    print("Maximum of a or b is " +max)

    // As expression
    // val max = if (a > b) a else b
}
```

The above piece of code yields the following output as a result in the browser. Our example also contains another line of code, which depicts how to use **“If”** statement as an expression.

```
Maximum of a or b is 5
```

Use of When

If you are familiar with other programming languages, then you might have heard of the term switch statement, which is basically a conditional operator when multiple conditions can be applied o

particular variable. **“when”** operator matches the variable value against the branch conditions. If it is satisfying the branch condition then it will execute the statement inside that scope. In the following example, we will learn more about “when” in Kotlin.

```
fun main(args: Array<String>) {
   val x:Int = 5
   when (x) {
      1 -> print("x = = 1")
      2 -> print("x = = 2")

      else -> { // Note the block
         print("x is neither 1 nor 2")
      }
   }
}
```

The above piece of code yields the following output in the browser.

```
x is neither 1 nor 2
```

In the above example, Kotlin compiler matches the value of **x** with the given branches. If it is not matching any of the branches, then it will execute the else part. Practically, when is equivalent to multiple if block. Kotlin provides another flexibility to the developer, where the developer can provide multiple checks in the same line by providing “,” inside the checks. Let us modify the above example as follows.

```
fun main(args: Array<String>) {
   val x:Int = 5
   when (x) {
      1,2 -> print(" Value of X either 1,2")

      else -> { // Note the block
         print("x is neither 1 nor 2")
      }
   }
}
```

Run the same in the browser, which will yield the following output in the browser.

```
x is neither 1 nor 2
```

For Loop

Loop is such an invention that provides the flexibility to iterate through any kind of data structure. Like other programming languages, Kotlin also provides many kinds of Looping methodolc

however, among them **“For”** is the most successful one. The implementation and use of For loop is conceptually similar to Java for loop. The following example shows how we can use the same in real-life examples.

```
fun main(args: Array<String>) {
   val items = listOf(1, 2, 3, 4)
   for (i in items) println("values of the array"+i)
}
```

In the above piece of code, we have declared one list named as “items” and using for loop we are iterating through that defined list and printing its value in the browser. Following is the output.

```
values of the array1
values of the array2
values of the array3
values of the array4
```

Following is another example of code, where we are using some library function to make our development work easier than ever before.

```
fun main(args: Array<String>) {
   val items = listOf(1, 22, 83, 4)

   for ((index, value) in items.withIndex()) {
      println("the element at $index is $value")
   }
}
```

Once we compile and execute the above piece of code in our coding ground, it will yield the following output in the browser.

```
the element at 0 is 1
the element at 1 is 22
the element at 2 is 83
the element at 3 is 4
```

While Loop and Do-While Loop

While and Do-While work exactly in a similar way like they do in other programming languages. The only difference between these two loops is, in case of Do-while loop the condition will be tested at the end of the loop. The following example shows the usage of the **While loop**.

```
fun main(args: Array<String>) {
   var x:Int = 0
   println("Example of While Loop--")
```

```
    while(x< = 10) {
       println(x)
       x++
    }
 }
```

The above piece of code yields the following output in the browser.

```
Example of While Loop--
0
1
2
3
4
5
6
7
8
9
10
```

Kotlin also has another loop called Do-While loop, where the loop body will be executed once, only then the condition will be checked. The following example shows the usage of the **Do-while loop**.

```
 fun main(args: Array<String>) {
    var x:Int = 0
    do {
       x = x + 10
       println("I am inside Do block---"+x)
    } while(x <= 50)
 }
```

The above piece of code yields the following output in the browser. In the above code, Kotlin compiler will execute the DO block, then it will go for condition checking in while block.

```
I am inside Do block---10
I am inside Do block---20
I am inside Do block---30
I am inside Do block---40
I am inside Do block---50
I am inside Do block---60
```

Use of Return, Break, Continue

If you are familiar with any programming language, then you must have an idea of different keywords that help us implement good control flow in the application. Following are the different keywords that can be used to control the loops or any other types of control flow.

Return − Return is a keyword that returns some value to the calling function from the called function. In the following example, we will implement this scenario using our Kotlin coding ground.

```
fun main(args: Array<String>) {
   var x:Int = 10
   println("The value of X is--"+doubleMe(x))
}
fun doubleMe(x:Int):Int {
   return 2*x;
}
```

In the above piece of code, we are calling another function and multiplying the input with 2, and returning the resultant value to the called function that is our main function. Kotlin defines the function in a different manner that we will look at in a subsequent chapter. For now, it is enough to understand that the above code will generate the following output in the browser.

```
The value of X is--20
```

Continue & Break − Continue and break are the most vital part of a logical problem. The “break” keyword terminates the controller flow if some condition has failed and “continue” does the opposite. All this operation happens with immediate visibility. Kotlin is smarter than other programming languages, wherein the developer can apply more than one label as visibility. The following piece of code shows how we are implementing this label in Kotlin.

```
fun main(args: Array<String>) {
   println("Example of Break and Continue")
   myLabel@ for(x in 1..10) { // appling the custom label
      if(x = = 5) {
         println("I am inside if block with value"+x+"\n-- hence it will close the
operation")
         break @myLabel //specifying the label
      } else {
         println("I am inside else block with value"+x)
         continue @myLabel
      }
   }
}
```

The above piece of code yields the following output in the browser.

```
Example of Break and Continue
I am inside else block with value1
I am inside else block with value2
I am inside else block with value3
I am inside else block with value4
```

```
I am inside if block with value5
-- hence it will close the operation
```

As you can see, the controller continues the loop, until and unless the value of **x** is 5. Once the value of **x** reaches 5, it starts executing the if block and once the break statement is reached, the entire control flow terminates the program execution.

Kotlin - Class & Object

In this chapter, we will learn the basics of Object-Oriented Programming (OOP) using Kotlin. We will learn about class and its object and how to play with that object. By definition of OOP, a class is a blueprint of a runtime entity and object is its state, which includes both its behavior and state. In Kotlin, class declaration consists of a class header and a class body surrounded by curly braces, similar to Java.

```
Class myClass { // class Header

   // class Body
}
```

Like Java, Kotlin also allows to create several objects of a class and you are free to include its class members and functions. We can control the visibility of the class members variables using different keywords that we will learn in Chapter 10 – Visibility Control. In the following example, we will create one class and its object through which we will access different data members of that class.

```
class myClass {
   // property (data member)
private var name: String = "mohamedsohel.co.in"

   // member function
fun printMe() {
      print("You are at the best Learning website Named-"+name)
   }
}
fun main(args: Array<String>) {
   val obj = myClass() // create obj object of myClass class
   obj.printMe()
}
```

The above piece of code will yield the following output in the browser, where we are calling printMe() of myClass using its own object.

```
You are at the best Learning website Named- mohamedsohel.co.in
```

Nested Class

By definition, when a class has been created inside another class, then it is called as a nested class. In Kotlin, nested class is by default static, hence, it can be accessed without creating any object of that class. In the following example, we will see how Kotlin interprets our nested class.

```
fun main(args: Array<String>) {
   val demo = Outer.Nested().foo() // calling nested class method
   print(demo)
}
class Outer {
   class Nested {
      fun foo() = "Welcome to The TutorialsPoint.com"
   }
}
```

The above piece of code will yield the following output in the browser.

```
Welcome to The TutorialsPoint.com
```

Inner Class

When a nested class is marked as a “inner”, then it will be called as an Inner class. An inner class can be accessed by the data member of the outer class. In the following example, we will be accessing the data member of the outer class.

```
fun main(args: Array<String>) {
   val demo = Outer().Nested().foo() // calling nested class method
   print(demo)
}
class Outer {
   private val welcomeMessage: String = "Welcome to the TutorialsPoint.com"
   inner class Nested {
      fun foo() = welcomeMessage
   }
}
```

The above piece of code will yield the following output in the browser, where we are calling the nested class using the default constructor provided by Kotlin compilers at the time of compiling.

```
Welcome to the TutorialsPoint.com
```

Anonymous Inner Class

Anonymous inner class is a pretty good concept that makes the life of a programmer very easy. Whenever we are implementing an interface, the concept of anonymous inner block comes i

picture. The concept of creating an object of interface using runtime object reference is known as anonymous class. In the following example, we will create an interface and we will create an object of that interface using Anonymous Inner class mechanism.

```
fun main(args: Array<String>) {
   var programmer :Human = object:Human // creating an instance of the interface {
      override fun think() { // overriding the think method
         print("I am an example of Anonymous Inner Class ")
      }
   }
   programmer.think()
}
interface Human {
   fun think()
}
```

The above piece of code will yield the following output in the browser.

```
I am an example of Anonymous Inner Class
```

Type Aliases

Type aliases are a property of Kotlin compiler. It provides the flexibility of creating a new name of an existing type, it does not create a new type. If the type name is too long, you can easily introduce a shorter name and use the same for future usage. Type aliases is really helpful for complex type. In the latest version, Kotlin revoked the support for type aliases, however, if you are using an old version of Kotlin you may have use it like the following −

```
typealias NodeSet = Set<Network.Node>
typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

Kotlin - Constructors

In this chapter, we will learn about constructors in Kotlin. Kotlin has two types of constructor - one is the **primary constructor** and the other is the **secondary constructor**. One Kotlin class can have one primary constructor, and one or more secondary constructor. Java constructor initializes the member variables, however, in Kotlin the primary constructor initializes the class, whereas the secondary constructor helps to include some extra logic while initializing the same. The primary constructor can be declared at class header level as shown in the following example.

```
class Person(val firstName: String, var age: Int) {
   // class body
}
```

In the above example, we have declared the primary constructor inside the parenthesis. Among the two fields, first name is read-only as it is declared as “val”, while the field age can be edited. In the following example, we will use the primary constructor.

```
fun main(args: Array<String>) {
   val person1 = Person("TutorialsPoint.com", 15)
   println("First Name = ${person1.firstName}")
   println("Age = ${person1.age}")
}
class Person(val firstName: String, var age: Int) {
}
```

The above piece of code will automatically initialize the two variables and provide the following output in the browser.

```
First Name = TutorialsPoint.com
Age = 15
```

As mentioned earlier, Kotlin allows to create one or more secondary constructors for your class. This secondary constructor is created using the “constructor” keyword. It is required whenever you want to create more than one constructor in Kotlin or whenever you want to include more logic in the primary constructor and you cannot do that because the primary constructor may be called by some other class. Take a look at the following example, where we have created a secondary constructor and are using the above example to implement the same.

```
fun main(args: Array<String>) {
   val HUman = HUman("TutorialsPoint.com", 25)
   print("${HUman.message}"+"${HUman.firstName}"+
      "Welcome to the example of Secondary  constructor, Your Age is-${HUman.age}")
}
class HUman(val firstName: String, var age: Int) {
   val message:String  = "Hey!!!"
        constructor(name : String , age :Int ,message :String):this(name,age) {
   }
}
```

Note – Any number of secondary constructors can be created, however, all of those constructors should call the primary constructor directly or indirectly.

The above piece of code will yield the following output in the browser.

```
Hey!!! TutorialsPoint.comWelcome to the example of Secondary constructor, Your Age is-
25
```

Kotlin - Inheritance

In this chapter, we will learn about inheritance. By definition, we all know that inheritance means accruing some properties of the mother class into the child class. In Kotlin, the base class is named as “Any”, which is the super class of the ‘any’ default class declared in Kotlin. Like all other OOPS, Kotlin also provides this functionality using one keyword known as **“:”**.

Everything in Kotlin is by default final, hence, we need to use the keyword “open” in front of the class declaration to make it allowable to inherit. Take a look at the following example of inheritance.

```
import java.util.Arrays

open class ABC {
   fun think () {
      print("Hey!! i am thiking ")
   }
}
class BCD: ABC(){ // inheritence happend using default constructor
}

fun main(args: Array<String>) {
   var  a = BCD()
   a.think()
}
```

The above piece of code will yield the following output in the browser.

```
Hey!! i am thiking
```

Now, what if we want to override the think() method in the child class. Then, we need to consider the following example where we are creating two classes and override one of its function into the child class.

```
import java.util.Arrays

open class ABC {
   open fun think () {
      print("Hey!! i am thinking ")
   }
}
class BCD: ABC() { // inheritance happens using default constructor
   override fun think() {
      print("I Am from Child")
   }
}
```

```
fun main(args: Array<String>) {
   var  a = BCD()
   a.think()
}
```

The above piece of code will call the child class inherited method and it will yield the following output in the browser. Like Java, Kotlin too doesn't allow multiple inheritances.

```
I Am from Child
```

Kotlin - Interface

In this chapter, we will learn about the interface in Kotlin. In Kotlin, the interface works exactly similar to Java 8, which means they can contain method implementation as well as abstract methods declaration. An interface can be implemented by a class in order to use its defined functionality. We have already introduced an example with an interface in Chapter 6 - section "anonymous inner class". In this chapter, we will learn more about it. The keyword "interface" is used to define an interface in Kotlin as shown in the following piece of code.

```
interface ExampleInterface {
   var myVar: String     // abstract property
   fun absMethod()       // abstract method
   fun sayHello() = "Hello there" // method with default implementation
}
```

In the above example, we have created one interface named as "ExampleInterface" and inside that we have a couple of abstract properties and methods all together. Look at the function named "sayHello()", which is an implemented method.

In the following example, we will be implementing the above interface in a class.

```
interface ExampleInterface  {
   var myVar: Int            // abstract property
   fun absMethod():String    // abstract method

   fun hello() {
      println("Hello there, Welcome to TutorialsPoint.Com!")
   }
}
class InterfaceImp : ExampleInterface {
   override var myVar: Int = 25
   override fun absMethod() = "Happy Learning "
}
fun main(args: Array<String>) {
```

```
    val obj = InterfaceImp()
    println("My Variable Value is = ${obj.myVar}")
    print("Calling hello(): ")
    obj.hello()

    print("Message from the Website-- ")
    println(obj.absMethod())
}
```

The above piece of code will yield the following output in the browser.

```
My Variable Value is = 25
Calling hello(): Hello there, Welcome to TutorialsPoint.Com!
Message from the Website-- Happy Learning
```

As mentioned earlier, Kotlin doesn't support multiple inheritances, however, the same thing can be achieved by implementing more than two interfaces at a time.

In the following example, we will create two interfaces and later we will implement both the interfaces into a class.

```
interface A {
    fun printMe() {
        println(" method of interface A")
    }
}
interface B  {
    fun printMeToo() {
        println("I am another Method from interface B")
    }
}

// implements two interfaces A and B
class multipleInterfaceExample: A, B

fun main(args: Array<String>) {
    val obj = multipleInterfaceExample()
    obj.printMe()
    obj.printMeToo()
}
```

In the above example, we have created two sample interfaces A, B and in the class named "multipleInterfaceExample" we have implemented two interfaces declared earlier. The above piece of code will yield the following output in the browser.

```
method of interface A
I am another Method from interface B
```

Kotlin - Visibility Control

In this chapter, we will learn about different modifiers available in Kotlin language. **Access modifier** is used to restrict the usage of the variables, methods and class used in the application. Like other OOP programming language, this modifier is applicable at multiple places such as in the class header or method declaration. There are four access modifiers available in Kotlin.

Private

The classes, methods, and packages can be declared with a private modifier. Once anything is declared as private, then it will be accessible within its immediate scope. For instance, a private package can be accessible within that specific file. A private class or interface can be accessible only by its data members, etc.

```
private class privateExample {
   private val i = 1
   private val doSomething() {
   }
}
```

In the above example, the class **"privateExample"** and the variable i both can be accessible only in the same Kotlin file, where its mentioned as they all are declared as private in the declaration block.

Protected

Protected is another access modifier for Kotlin, which is currently not available for top level declaration like any package cannot be protected. A protected class or interface is visible to its subclass only.

```
class A() {
   protected val i = 1
}
class B : A() {
   fun getValue() : Int {
      return i
   }
}
```

In the above example, the variable **"i"** is declared as protected, hence, it is only visible to its subclass.

Internal

Internal is a newly added modifier introduced in Kotlin. If anything is marked as internal, then that specific field will be in the internal field. An Internal package is visible only inside the module under which it is implemented. An internal class interface is visible only by other class present inside the same package or the module. In the following example, we will see how to implement an internal method.

```
class internalExample {
   internal val i = 1
   internal fun doSomething() {
   }
}
```

In the above example, the method named “doSomething” and the variable is mentioned as internal, hence, these two fields can be accessible only inside the package under which it is declared.

Public

Public modifier is accessible from anywhere in the project workspace. If no access modifier is specified, then by default it will be in the public scope. In all our previous examples, we have not mentioned any modifier, hence, all of them are in the public scope. Following is an example to understand more on how to declare a public variable or method.

```
class publicExample {
   val i = 1
   fun doSomething() {
   }
}
```

In the above example, we have not mentioned any modifier, thus all these methods and variables are by default public.

Kotlin - Functions

Kotlin is a statically typed language, hence, functions play a great role in it. We are pretty familiar with function, as we are using function throughout the examples. Function is declared with the keyword “fun”. Like any other OOP, it also needs a return type and an option argument list.

In the following example, we are defining a function called MyFunction and from the main function we are calling this function and passing some argument.

```
fun main(args: Array<String>) {
   println(MyFunction("tutorialsPoint.com"))
}
fun MyFunction(x: String): String {
   var c:String  = "Hey!! Welcome To ---"
   return (c+x)
}
```

The above piece of code will yield the following output in the browser.

```
Hey!! Welcome To ---tutorialsPoint.com
```

The function should be declared as follows −

```
fun <nameOfFunction>(<argument>:<argumentType>):<ReturnType>
```

Following are some of the different types of function available in Kotlin.

Lambda Function

Lambda is a high level function that drastically reduces the boiler plate code while declaring a function and defining the same. Kotlin allows you to define your own lambda. In Kotlin, you can declare your lambda and pass that lambda to a function.

Take a look at the following example.

```
fun main(args: Array<String>) {
   val mylambda :(String)->Unit  = {s:String->print(s)}
   val v:String = "TutorialsPoint.com"
   mylambda(v)
}
```

In the above code, we have created our own lambda known as “mylambda” and we have passed one variable to this lambda, which is of type String and contains a value “TutorialsPoint.com”.

The above piece of code will yield the following output in the browser.

```
TutorialsPoint.com
```

Inline Function

The above example shows the basic of the lambda expression that we can use in Kotlin application. Now, we can pass a lambda to another function to get our output which makes the calling funct

an inline function.

Take a look at the following example.

```
fun main(args: Array<String>) {
   val mylambda:(String)->Unit  = {s:String->print(s)}
   val v:String = "TutorialsPoint.com"
   myFun(v,mylambda) //passing lambda as a parameter of another function
}
fun myFun(a :String, action: (String)->Unit) { //passing lambda
   print("Heyyy!!!")
   action(a)// call to lambda function
}
```

The above piece of code will yield the following output in the browser. Using inline function, we have passed a lambda as a parameter. Any other function can be made an inline function using the “inline” keyword.

```
Heyyy!!!TutorialsPoint.com
```

Kotlin - Destructuring Declarations

Kotlin contains many features of other programming languages. It allows you to declare multiple variables at once. This technique is called Destructuring declaration.

Following is the basic syntax of the destructuring declaration.

```
val (name, age) = person
```

In the above syntax, we have created an object and defined all of them together in a single statement. Later, we can use them as follows.

```
println(name)
println(age)
```

Now, let us see how we can use the same in our real-life application. Consider the following example where we are creating one Student class with some attributes and later we will be using them to print the object values.

```
fun main(args: Array<String>) {
   val s = Student("TutorialsPoint.com","Kotlin")
   val (name,subject) = s
   println("You are learning "+subject+" from "+name)
}
data class Student( val a :String,val b: String ){
```

```
    var name:String = a
    var subject:String = b
}
```

The above piece of code will yield the following output in the browser.

```
You are learning Kotlin from TutorialsPoint.com
```

Kotlin - Exception Handling

Exception handling is a very important part of a programming language. This technique restricts our application from generating the wrong output at runtime. In this chapter, we will learn how to handle runtime exception in Kotlin. The exceptions in Kotlin is pretty similar to the exceptions in Java. All the exceptions are descendants of the “Throwable” class. Following example shows how to use exception handling technique in Kotlin.

```
fun main(args: Array<String>) {
    try {
        val myVar:Int = 12;
        val v:String = "Tutorialspoint.com";
        v.toInt();
    } catch(e:Exception) {
        e.printStackTrace();
    } finally {
        println("Exception Handeling in Kotlin");
    }
}
```

In the above piece of code, we have declared a String and later tied that string into the integer, which is actually a runtime exception. Hence, we will get the following output in the browser.

```
val myVar:Int = 12;
Exception Handeling in Kotlin
```

Note − Like Java, Kotlin also executes the finally block after executing the catch block.